In-Storage Embedded Accelerator for Sparse Pattern Processing

Sang-Woo Jun[*], Huy T. Nguyen[#], Vijay Gadepally[#*], and Arvind[*]
[#]MIT Lincoln Laboratory, [*]MIT Computer Science & Artificial Intelligence Laboratory

Abstract— **We present a novel system architecture for sparse pattern processing, using flash storage and an in-storage embedded accelerator. Placing commonly used computing kernels in direct access to a data source achieves high performance, without increasing the requirements for system memory. We show that the sparse pattern matching accelerator is useful for general sparse vector multiplication, feature matching, subgraph matching, protein database search, and machine learning applications. In our prototyping experiment, one accelerator slice can outperform a 16-core system at a fraction of the power and cost.**

I. Introduction

Many data analytics of interest deal with sparse data, including audio, video, images, and documents. Document search or clustering algorithms often represent documents as a bag-of-words, and many algorithms, including important machine learning algorithms, have been designed to work on data represented as sparse matrices and vectors.

In many cases, the size of data is too large to be accommodated in the system memory of a single machine [1]. At the same time, the processing requirement of the problems is large enough that processing itself can become the bottleneck. The traditional way to overcome this situation is to either use a cluster of machines so that data can be accommodated in the collective main (DRAM) memory, and computation can be distributed across the machines in the cluster [2]. Scaling out, however, increases cost and reduces efficiency. Recently, new server systems can accommodate large amounts of memory and multiple CPU sockets, enabling more memory and compute capacity per platform. A quad-socket system with 1.5 TB memory and four 12-core CPUs (48 cores total) can be put together for $27k, or with high premium 18-core CPUs for $49k. This is still a costly solution. The cost increases with memory and very steeply with number of cores in the system.

One alternative solution is to use a fast flash-based secondary storage such as Solid-State Drives (SSDs) for the main data store instead of a large memory, accepting that most of data access will happen on the secondary storage, and optimize for it. Industry has been heading this way with recent advances such as Non-Volatile Memory Express (NVMe) devices, making this type of solution more attractive.

On the computing front, using application-specific hardware accelerators [3] can offer one to three orders of magnitude better performance with less power consumption compared to CPU cores performing the same task. Many accelerator devices are packaged as an independent system component that can be plugged into a high-speed bus such as PCIe to interface with CPUs and system memory.

This work is sponsored by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract number FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

In this paper, we demonstrate a novel system architecture called In-Storage Computing that integrates the storage and computing solutions together to reduce memory size and CPU workload [4]. An illustration is shown in Figure 1. The Field-Programmable Gate Array (FPGA) is used as the application-specific accelerator. In this architecture, it takes on two additional roles: to interface directly with storage, and to provide dedicated high bandwidth network for scalability.

In high-volume data processing, besides the number crunching tasks, there are many other activities such as handling data structures, indexing, pointer arithmetic, file operation, network operation, etc., that also demand significant CPU resources. Most could be off-loaded to very efficient application-specific logic in the FPGA.

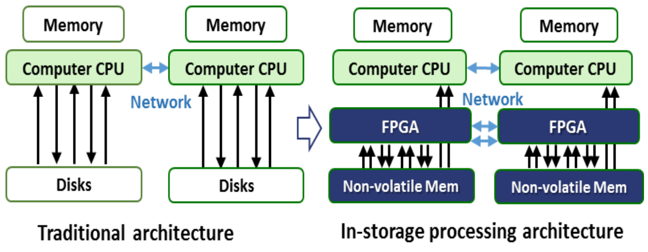


Figure 1: In-Storage Computing Architecture

Sparse pattern processing is one class of applications that requires complex data handling and does not lend itself to the pipelined SIMD vector processing units in the CPU. This is one good candidate for acceleration on our in-storage computing architecture. Our particular application is document search, which can be described as: Given a document with one set of prominent words, search for others that may cover similar topic(s) or sentiment(s). The words are sparse in vocabulary space, hence, sparse pattern processing. The operations in sparse pattern processing can be extended to natural language processing, bioinformatics, subgraph matching, machine learning, and graph processing. [5][6][7]. A recent effort, the GraphBLAS [8], aims to standardize some of these computational kernels in order to support the development of hardware-acceleration [9].

A well-designed accelerator could benefit many fields. We will show that our baseline accelerator outperforms a 16-core large memory server system, and matches a 24-core system under certain conditions, using only 2/3 power. An optimized version could match a 48-core system at ¼ power and cost.

The paper is organized as follows. Section II describes sparse pattern matching using document search as an example. Section III presents some important applications that could benefit from accelerated sparse pattern matching. Section IV introduces our architecture in detail. Section V describes implementation details of the prototype system and its performance measurements. Sections VI and VII conclude with discussions and summary.

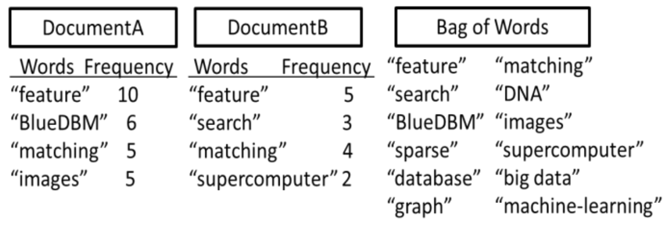
II. Sparse Pattern Matching

Many important types of information can be naturally represented in a sparse manner, and many important applications deal with data organized into a sparse representation, such as sparse vector or matrices. A high-performance platform for processing sparse patterns can be a useful tool for solving many problems. In this section, we describe sparse pattern matching using one of its key applications, document matching.

A. *Document Matching*

The objective of document matching is to find document candidate(s) that match best to a query document [10][11][12]. Documents that discuss similar topics would use similar vocabulary at high level of occurrences. The topics can be abstracted out into mathematical models based on the words, their frequencies, and with more sophistication, the ordering and grouping of the word appearances. Example uses are in natural language understanding, relationship extraction, sentiment analysis, topic segmentation, information retrieval, predictive analysis, and bioinformatics.

A simple representation of a document search application is shown in Figure 2. Documents A and B would be preprocessed to extract words with occurrences above some set threshold. The vocabulary set is the superset of all prominent words in all documents in the collection. In the UCI Machine Learning Repository [13], the collection of NY Times articles contains around 300,000 documents with about 100,000,000 words and a prominent vocabulary set of about 100,000. The Enron email collection has almost 40,000 emails with about 28,000 prominent words.



DocumentA

Words	Frequency
"feature"	10
"BlueDBM"	6
"matching"	5
"images"	5

DocumentB

Words	Frequency
"feature"	5
"search"	3
"matching"	4
"supercomputer"	2

Bag of Words

"feature"	"matching"
"search"	"DNA"
"BlueDBM"	"images"
"sparse"	"supercomputer"
"database"	"big data"
"graph"	"machine-learning"

Figure 2: Document Matching

B. *Comparison metric: Cosine similarity*

Each document can be represented as an N-dimensional vector, where N is the size of the bag-of-words. In the above example, the Document-A vector is stored as a sparse vector of 4 non-zero elements. Document-B vector also happens to have 4 non-zero elements although this could vary widely depending the number of prominent words in the document above the set threshold.

One simple and effective method for comparing high-dimensional vectors is the *Cosine similarity metric*, which is defined as follows (the top computes the correlation, and the bottom performs the normalization):

$$\cos(\theta) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

As illustrated in Figure 3, the vectors that are closely aligned would result in large Cosine metric.

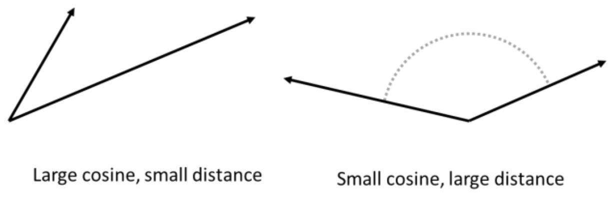


Figure 3: Cosine as Alignment Metric

The normalization in the above equation can be pre-computed for each vector. Let's now focus on the numerator, which computes the correlation, or dot product, of vectors A and B. This computation is illustrated in Figure 4 for both sparse and full formats. The term-wise multiplication of A_i and B_i creates partial products PP_i for each term, but due to sparsity, only terms PP_1 and PP_7 need to be created. The correlation score is the summation of these partial products.

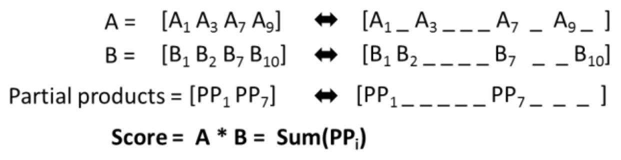

A = [A1 A3 A7 A9] ⬌ [A1 _ A3 _ _ _ A7 _ A9 _]
B = [B1 B2 B7 B10] ⬌ [B1 B2 _ _ _ _ B7 _ _ B10]
Partial products = [PP1 PP7] ⬌ [PP1 _ _ _ _ _ PP7 _ _ _]
Score = A * B = Sum(PPi)

Figure 4: Pattern Matching as Sparse Vector Multiplication

The match processing between a document-A against the whole collection can be formulated as a matrix-vector multiplication as illustrated in Figure 5, where both the matrix and the vector are sparse. If the queries are batched together, for example, [document-AA, document-AB, …], the problem can be formulated as sparse matrix-matrix multiplication.

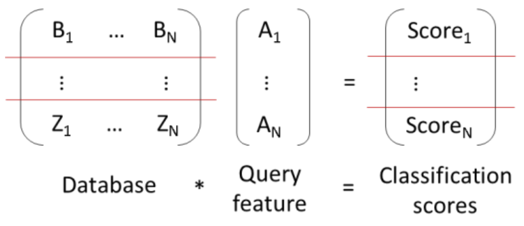


Figure 5: Document Search and Classification

There are many opportunities for parallelizing the processing. As illustrated in Figure 5, one approach is to partition the database matrix into K subsets (separated by dividing lines) and compute in parallel on K accelerator kernels. If a batch of L queries is issued, the work can be performed in parallel on $K*L$ kernels. Analysis is required to determine the value of K and L parameters to match optimally the accelerator computing capability, data bandwidth, and local working storage.

III. Other Applications

A. *Subgraph Matching*

Document matching could be extended into the graph processing domain by considering subgraphs to be documents, and conducting a search for subgraphs that contain similar vertices, as illustrated in Figure 6. Each edge is mapped into a "word" that contains the labels of its two vertices, and subgraphs can have different number of edges. The labeling and partitioning into subgraphs depends on the application context, our focus here is on the acceleration of subgraphs matching.

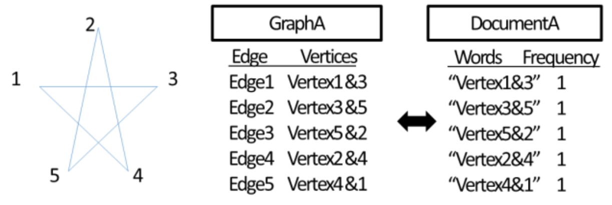


Figure 6: Subgraph Matching

Each edge traversed in the graph corresponds to a word match in the document comparison, or equivalently, a non-zero partial product.

B. *Feature Matching in Machine Learning*

Machine learning algorithms train their classifiers by churning through massive data to evaluate "feature scores", and attempt to optimize an objective function that aligns the classifications to desired outcomes. This involves adjusting the coefficients and repeatedly going through the evaluation process until the accuracy is deemed acceptable.

For example, a facial recognition application would operate on features extracted from facial images. A feature vector corresponds to a document, and the training process involves many rounds of matching documents against classifier candidates, not too different from scenario depicted in Figure 5.

Operation on sparse features further enables efficient evaluation of neural networks. Most approaches based on neural nets require a large training dataset and intense number crunching. If these requirements can be met for an application in a low-power and portable form factor, the training of neural networks could take place in the field for a variety of different applications.

C. *Bioinformatics*

High dimensional search using a bag-of-words representation is also useful in the field of bioinformatics. For example, researchers who discovered new protein sequences use protein search tools such as BLAST [14] to find previously discovered and annotated proteins in order to guess the genealogy and function of the newly discovered protein. Since performing an optimal search algorithm such as Smith-Waterman on the entire annotated protein database is expensive, each protein sequence in the database is pre-processed into a more easily searchable format, such as bag-of-words. The goal of this pre-processed format is not to find the most similar reference sequence by itself, but rather to determine a smaller set of reference sequences that are statistically likely to be similar to the query. This reduces the search space of the optimal algorithm.

We have experimented with this approach, by first pre-process each protein sequence to generate a sparse bag-of-words representation of all 3-mers, and then perform a global search. The sparse metadata extracted from the UniProt TrEMBL dataset [15] is reduced from 35 GBs to approximately 4 GBs, and requires only 2 seconds in our flash-based prototype to be traversed in its entirety.

Figure 7 shows how an example protein sequence can be organized into a bag-of-words representation. Once all reference protein sequences in the dataset are processed and encoded in such a sparse format, the same document search kernel can be used to quickly search for reference proteins statistically likely to be similar to the query.

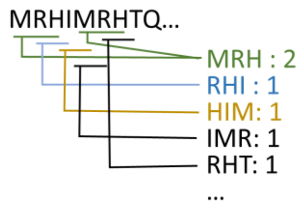


Figure 7: Organizing a Protein Sequence into Bag-of-Words

IV. Accelerator Design

As part of the design process, we performed analysis and experiments to gain insights into the application and identify bottlenecks. Decomposing the application into key functions and performing benchmarking, we can tell if performance is bounded by data bandwidth or computing capability.

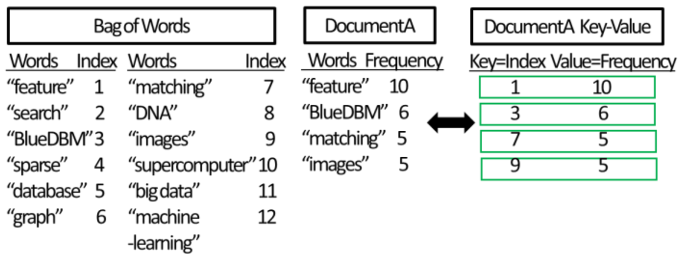
For document matching, computing the Cosine similarity metric is the key function and specialized hardware, in the form of sparse vector multiplication, can have a significant effect on the performance. This application can also use multiple copies of this accelerator in parallel.

Fast data access is almost always required to keep the computing kernels busy. Each vector multiplication requires a new vector to be delivered to the kernel. On a system with large memory, all vectors can be loaded and kept in memory at first use so that subsequent accesses are fast. But on smaller system, secondary storage will need to be accessed, thus, limiting the data bandwidth to the kernels. In the current technology, there is a 25X to 100X difference in system memory bandwidth and the secondary storage bandwidth.

Whereas a CPU-based system with high memory bandwidth is likely to be compute bound, our accelerator using flash would be bandwidth bound. It is, therefore, critical that we use data structures that are as bandwidth efficient as possible.

A. *Data Structure*

Our document encoding schema is illustrated in Figure 8. Rather than storing the words, we are storing the index into the bag-of-words. This requires an indexing step, but allows for much flexibility in abstraction, for example, "words" can be phrases, strings, or even other features (images) of the document, etc.



Bag of Words

Words	Index	Words	Index
"feature"	1	"matching"	7
"search"	2	"DNA"	8
"BlueDBM"	3	"images"	9
"sparse"	4	"supercomputer"	10
"database"	5	"big data"	11
"graph"	6	"machine -learning"	12

DocumentA

Words	Frequency
"feature"	10
"BlueDBM"	6
"matching"	5
"images"	5

DocumentA Key-Value

Key=Index	Value=Frequency
1	10
3	6
7	5
9	5

Figure 8: Document Encoding Schema

To make effective use of storage bandwidth and computation, we encoded the vectors, or pattern datasets, into binary format as shown in Figure 9. Each data item is 32 bits wide. Pending on a flag bit, each data item can either be a pattern identifier, i.e., "Document A", or a key/value pair for each word in the document. Comparing to the original data format from UCI repository [18], which replicates the documentID with each wordID and storing one word per line of 256 bytes, this format uses only one 4-byte documentID followed words encoded in 4 bytes each. This compact format significantly saves storage bandwidth, considering each document contains about 60 prominent words.

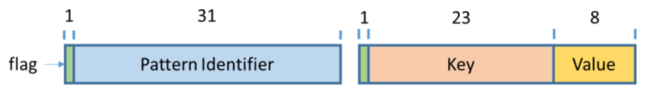


Figure 9: Data Format

B. *Architecture*

A reference software implementation was developed to provide benchmark functionality and a framework for later integration of the accelerator. The implementation is parameterized so the user can choose the number of worker threads to spawn. Each worker is given a contiguous partition of the dataset to work on, along with its own copy of the query pattern. After finishing execution, each thread reports its top pattern with the largest cosine metric. The main thread then selects the global nearest pattern from the small list of local patterns.

In the FPGA-accelerated design, the FPGA platform is inserted between the flash storage and the host server. A software scaffolding infrastructure [4] shown in Figure 13 is provided for rapid development of multiple accelerator kernels. Since most of actual computation is offloaded into the FPGA, the host software is only responsible for managing data organization and routing, and almost no actual computation. As a result, a very small processor could be used and still make full use of storage device bandwidth.

When the host software sends a read or write command to the storage device, the data transfer is default to be between storage to/from host server memory. Optionally, it can specify source and destination to be one of the various accelerator kernels on the FPGA. The host software also provides an additional interface for sending and receiving sideband information to each kernel.

Figure 10 shows the interaction of host software, accelerator, and flash storage. Each FPGA accelerator kernel implements the same interface, and can be accessed through 4 ports: dataIn (512 bits), commandIn (128 bits), resultToMemory (128 bits), and dataToStorage (512 bits). Data read from flash storage is streamed through dataIn, and the host software sends additional information such as commands via the sideband port commandIn. As the accelerator performs computation, it can send data that needs to be stored to flash via dataToStorage, or to host software via resultsToMemory. The accelerator can send data over these two ports at any time during computation. The host software is responsible for managing the ports, for example, maintaining a list of free pages that dataToStorage uses.

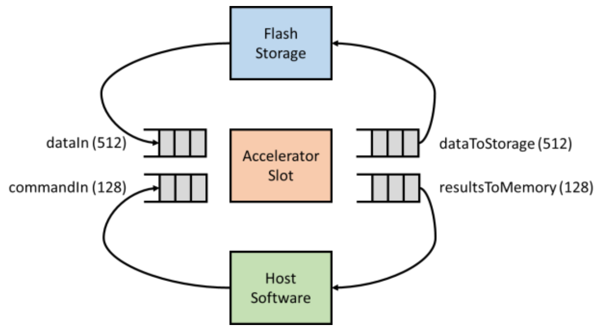


Figure 10: Accelerator Interface

An architectural view of the sparse pattern matching kernel is illustrated in Figure 11. It consists of a query memory, accessed through the sideband datapath, and a chain of computation modules for calculating the cosine similarity between the query and data from flash storage. Multiple kernels are implemented in the system in order to make full use of the flash storage bandwidth. Each kernel receives its own data routed from flash storage as controlled by the host software, which sends pages in round-robin fashion for best use of flash bandwidth and maximizing accelerator performance. Data pages from flash do not necessarily arrive in order, which necessitates re-ordering and other low-level coordination activities that take place in the flash storage interface logic block [4].

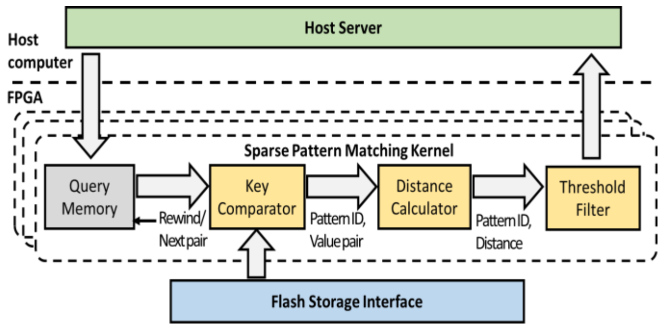


Figure 11: Sparse Matching Accelerator Architecture

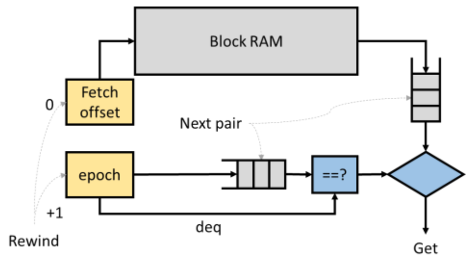


Figure 12: Prefetching of Query

In the matching of the two sparse encoded vectors, the accelerator kernel uses two pointers to query memory and data from flash that has been routed to block RAM as illustrated in Figure 12. These pointers track the key/value pair being processed. If the keys in both memories match, appropriate processing can take place. Otherwise, one of the pointers needs to be incremented depending on which is bigger. Unlike the data pointer, which is always increasing because the data is streaming in from flash, the query pointer needs to be "rewound" when the end of a vector is reached. But if we were to only load query data only after we knew no rewinding is required, we would waste valuable cycles and not be keeping the query memory busy. As a solution, a prefetch predictor has been added to the query memory block to queue prefetched values into the comparison unit in Figure 12. If it is determined that prefetched values are not needed (Rewind), they are flushed by the dequeuing logic.

V. Implementation & Results

A. System description

Our application was implemented on a single-node setup of MIT's BlueDBM system [10]. The single-node BlueDBM system consists of a 24-core Xeon server and a BlueDBM storage device, which is a pair of a Xilinx VC707 FPGA development board and custom flash modules with 1 TB capacity. The flash device is capable of 2 GB/sec throughput. A host server also includes 50 GBs of DRAM. The BlueDBM device and host server are connected via a Gen2 x8 PCIe link. Figure 13 highlights one "accelerator slice" on a server computer and the software infrastructure for integrating the accelerator.

Eight accelerator cores were used to fully saturate flash bandwidth. Each accelerator was given 8 KBs of query memory on the on-chip block RAM. The 8-KB size allows for up to 2K non-zero elements in the sparse query vector. The size of the query memory could be made larger and more accelerator cores could be used, but this was more than adequate for our example application.

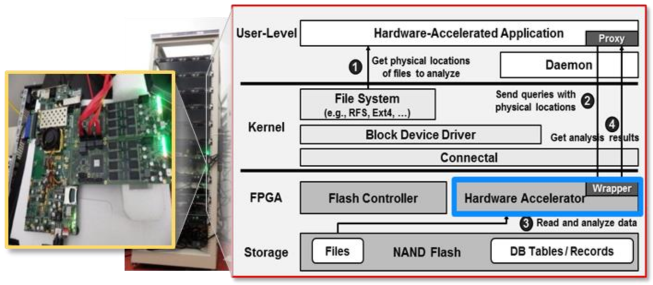


Figure 13: BlueDBM System

We used UCI's bag-of-words dataset [13] as the example dataset. The UCI dataset processed multiple document collections including Pubmed, and created bag-of-words databases of each collection. Each database consists of a vocabulary file that maps words to wordIDs, and a doc file that contains {documentID, wordID, wordCount} tuples. For effective use of disk and memory bandwidth, we encoded each dataset in the binary format described earlier in Figure 9. Since the UCI dataset is fairly small, we developed a data synthesizer that generates permutations to create datasets of 100 GBs or more for our experiments. The synthesizer permuted documents in the dataset by adding and removing random words, and assigning random word count values to some words.

B. Results

The document matching performance was averaged over an interval of 10 seconds for several system configurations: (1) Data on hard disk, (2) Data on "RAM-Disk", (3) Data stored in memory, and (4) BlueDBM with data in flash.

Configuration (2) bypasses the mechanical operation of the disk but still requires the computer operating system to perform file operations, which approximates the performance upper bound with SSD devices. Configuration (3) corresponds to the new trend of high-end in-memory database processing enabled by recent increase in computer memory sizes and substantially many more CPU cores available to take advantage of the large memory working set. Configuration (4) off-loads most operation to its FPGA accelerator, thus, places minimum requires on CPU and memory. The BlueDBM flash modules provide affordable large storage with high transfer rate to keep the accelerator operating at full throttle.

Table 1: Document Matching Rate

Number of CPU threads	CPU - Matlab	CPU – C++			BlueDBM with 8 comparators
	In memory	Data in hard drive	Data in file buffer	Data in memory	
1	0.31	0.31 (M)	0.88 (M)	0.94 (M)	NA
2	0.31	0.38	1.56	1.73	10.35 (M)
4	0.34	0.41	2.68	3.35	
8		0.44	4.15	6.64	
16		0.46	5.84	9.76	
24	0.6	0.5	6.43	13.17	

Table 1 shows millions of documents matched per second for the various configurations, scaling from 1 CPU threads to 24 threads. A graphical depiction is given in Figure 14. Configuration (1) with hard disk is clearly limited by the disk bandwidth. Configuration (2) scales up nicely, but eventually runs into limitation by file processing functions within the operating system, and peaks at 6 million docs/sec. This could be viewed as the upper bound for storage solution with RAIDed SSDs.

Configuration (3) benefits from its direct access to data in memory, achieving the highest rate of 13 million docs/sec. In real-life setting, the computer would need to read data from secondary storage into memory at least once, so performance would be somewhere in between configurations (2) and (3).

Configuration (4) reaches its peak at 10 million, limited by the bandwidth of BlueDBM flash modules. This is slightly lower than configuration (3), but almost twice configuration (2).

Note that configuration (4) achieves 10 million docs/sec with only 2 CPU threads, leaving the rest available for other functions. This also suggests that a much smaller CPU, such as those embedded in the FPGA could be used, hence, an interesting concept of shrinking and folding the host into the accelerator itself!

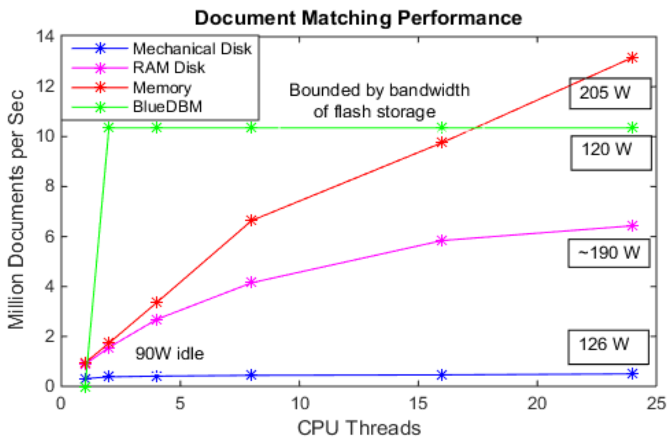


Figure 14: Document Processed vs CPU Threads

The power measurements are given in Table 2, ranging from 90 Watts idle to 205 Watts maximum by configuration (3). As labeled in Figure 14, configuration (4) draws nearly 40% less power - only 120 Watts while delivering a slight reduction in processing rate. This solution is clearly more power efficient than configuration (2) which draws more power but delivers much less processing.

Table 2: Power Dissipation

Number of CPU threads	CPU – C++		BlueDBM with 8 comparators
	Data in hard drive	Data in memory	
1	120	114	NA
2	110	134	120 W *(idle 90W)*
4	100	145	
8	123	160	
16	112	164	
24	126	205	

VI. Discussion

A. *Scalability & Cost*

The performance of BlueDBM accelerator is bounded by the bandwidth between the accelerator kernels and the flash storage. With 8 kernels processing 10 million docs/sec, the sustaining data rate was measured to be about 2 GB/sec. If the same volume of data could support more processing, such as when queries are issued in batched, then more kernels could be added to the accelerator. As depicted in Table 3, the FPGA could easily provide 20 kernels to churn through 27 million docs/sec if queries are issued in batch of 3. Alternately, this performance could also be achieved for single query case with an increase in flash storage bandwidth to 6 GB/sec, by either adding more parallelism to BlueDBM flash modules or upgrading the architecture to use PCIe-based modules presently available.

Table 3: Accelerator Scalability

Kernels in FPGA	Million Docs / sec	Flash IO (GB/s)
8	10.35	~2
20	27 est.	5.4 est.

The CPU in-memory processing, in contrast, is not bandwidth but rather processing limited. Using CPU with more cores would increase performance, of course, at a steep premium. Figure 15 projects that with 48-cores, a server system could achieve 27 million docs/sec. With today's technology, it is not possible to obtain a 48-core system in a dual-CPU setting, but a quad-CPU platform can meet this requirement with 4 12-core CPUs.

A quad-CPU 48-core system with ~1.5 TB of memory can be obtained from Colfax for $27k, with approximate break down of $10k for memory, $7k for platform and storage, and $10k for four mid-range 12-core CPUs ($32k for four high-end 18-core CPUs). In contrast, a FPGA-based accelerator slice including flash modules can be built from commercial components for less than $6k.

Our accelerator could be implemented in two ways: (1) with flash modules directly plugged into a FPGA card as in BlueDBM hardware, or (2) allowing the flash modules to be off-board but maintaining a tight interaction over PCIe, such as the upgraded version using NVMe flash devices. The first approach requires less PCIe slots on the platform and provides better dedicated data bandwidth, whereas the second allows for more scalability on large platform and possibly easier upgrade with latest technology.

B. *Power Optimization*

Figure 15 captures performance and power dissipation for configurations 3 and 4 discussed earlier. The solid red curve depicts CPU in-memory processing, and the solid green curve for BlueDBM in-storage processing. The dotted curves project the trend out to the capability of a slightly modified BlueDBM system that could be put together with today's technology as discussed in Table 3.

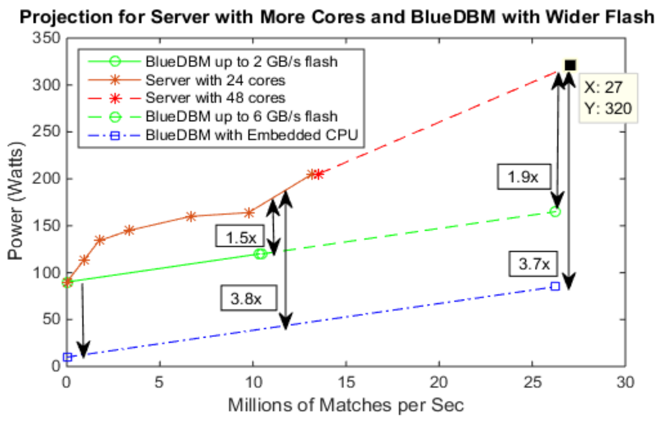


Figure 15: Projection to Technology Generations

The blue curve shows an interesting variant of BlueDBM where the host CPU is downsized and absorbed into the accelerator itself, i.e., the host-side 2-thread software now run on the embedded CPU within the FPGA. This presents a solution for extreme Size, Weight, and Power (SWaP) applications. The high density ratio of flash versus computer memory, and simplification in power conditioning circuitries and cooling, etc., would enable a compact implementation suitable for portable use cases.

The embedded CPU solution would draw only 10 Watts at idle rather than 90 Watts as in the full-up design, At the 10 million doc/sec performance point, the power dissipation is less than 50 Watts, and 27 million docs/sec requires only 90 Watts. This compact implementation would be about **3.8 times more power efficient** than CPU in-memory processing.

C. *Sparse Multiplications Performance*

Earlier in Section II, the sparse pattern matching problem processing was formulated as a multiplication of a sparse vector by a sparse vector in Figure 4. This allows us to project our performance to a generic metric of partial products per second, where a partial product is produced only when its two multiplicands are non-zeros, thus, resulting in a meaningful output.

Our processing of 8.2 million documents with 483 million words generates 11 million partial products in about 0.8 seconds. This yields a performance of **13 million partial products /sec** for sparse vector multiplication. The sparsity of the data is characterized by the number of "non-trivial" words out of 141,000 vocabulary set. On average, a document contains about 60 such words, hence, a sparsity of 60/141,000 = 0.04%.

We could notionally attempt to compare our partial product processing rate to Graphulo server-side sparse matrix multiplication [16], keeping in mind 1) our results are immediately consumed – no need for saving back to database, and 2) the two datasets even if close in sparsity level, will have different number of partial products due to the actual distribution of non-zero terms. The data in the report had 16 non-zeros per vector, and for a vector of size 2^{17} = 131,072 ~141,000, the sparsity comes out to be 0.01%, not too far from our application parameter.

Graphulo server-side multiplication peaks at about 300,000 partial products /sec per pair of CPU cores, attributed to architecture of the Tablet server. Assuming optimistic linear scaling with CPU cores, it would take a system with **43 CPU cores** to match one BlueDBM accelerator card.

VII. Summary

Sparse pattern processing is an important component that can be used in many applications dealing with a great amount of data, including text analysis, bioinformatics and machine learning. In this paper, we have presented a novel system architecture that uses high performance storage and in-storage accelerators for sparse pattern processing, and demonstrated its validity using a prototype implementation.

Sparse pattern matching techniques such as those described in the article have wide applicability to a variety of applications. For example, the subgraph-matching problem [17] is important in biological networks, event recognition and community detection. Other applications such as topic modeling [18], centrality [19], and graph traversal are also amenable to sparse processing techniques [5].

Our prototype accelerator streaming data from flash storage was able to outperform a 16-core multithreaded software implementation with all of the data stored in DRAM, while consuming much less power. A cluster of machines with such accelerators could deliver comparable performance compared to a larger conventional cluster, while costing much less to purchase and operate. The proposed architecture could be a desirable alternative to conventional computer architectures including cloud services and low-power field applications.

There are many directions for future work. First, we would like to explore the application of in-storage embedded accelerators in a cloud [20][21] or supercomputing setting. Our proposed system would be an ideal candidate to perform analysis of enterprise and research problems in such environments. We are also currently developing GraphBLAS compliant operations in our system in order to perform common graph and sparse linear algebra problems such as one proposed in [22]. We would also like to investigate how to integrate our system into more general data management solutions such as the BigDAWG polystore system [23].

REFERENCES

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51, no. 1 (2008): 107-113.

[2] Zaharia, Matei, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets." *HotCloud* 10 (2010): 10-10.

[3] Huy Nguyen, James Haupt, Michael Eskowitz, Birol Bekirov, Jonathan Scalera, Thomas Anderson, Michael Vai, and Kenneth Teitelbaum, “High-Performance FPGA-Based QR Decomposition,” High Performance Embedded Computing Workshop, 2005.

[4] Jun, Sang-Woo, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, and Shuotao Xu. "Bluedbm: an appliance for big data analytics." In*Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 1-13. IEEE, 2015.

[5] Gadepally, Vijay, Jake Bolewski, Dan Hook, Dylan Hutchison, Ben Miller, and Jeremy Kepner. "Graphulo: Linear algebra graph kernels for NoSQL databases." In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 822-830. IEEE, 2015.

[6] Eggert, Julian, and Edgar Körner. "Sparse coding and NMF." In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 4, pp. 2529-2533. IEEE, 2004.

[7] Smola, Alex J., and Bernhard Schölkopf. "Sparse greedy matrix approximation for machine learning." (2000).

[8] http://graphblas.org/

[9] Bader, David, Aydın Buluç, John Gilbert, Joseph Gonzalez, Jeremy Kepner, and Timothy Mattson. "The Graph BLAS effort and its implications for Exascale." In *SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities (EX14)*. 2014

[10] Mimno, David, and Andrew McCallum. "Expertise modeling for matching papers with reviewers." In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 500-509. ACM, 2007.

[11] Gavin, Brendan, Vijay Gadepally, and Jeremy Kepner. "Large Enforced Sparse Non-Negative Matrix Factorization." In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2016 IEEE International*, IEEE, 2016.

[12] Steinbach, Michael, George Karypis, and Vipin Kumar. "A comparison of document clustering techniques." In *KDD workshop on text mining*, vol. 400, no. 1, pp. 525-526. 2000.

[13] Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science

[14] Altschul, Stephen; Gish, Warren; Miller, Webb; Myers, Eugene; Lipman, David (1990). "Basic local alignment search tool". Journal of Molecular Biology 215 (3): 403–410.

[15] Bairoch A, and Apweiler R., The Swiss-Prot protein sequence data bank and its supplement TrEMBL in 2000, Nucl. Acids Res. 28:45-48(2000).

[16] Hutchison, Dylan, Jeremy Kepner, Vijay Gadepally, and Adam Fuchs. "Graphulo implementation of server-side sparse matrix multiply in the Accumulo database." In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pp. 1-7. IEEE, 2015.

[17] Sun, Zhao, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. "Efficient subgraph matching on billion node graphs." *Proceedings of the VLDB Endowment* 5, no. 9 (2012): 788-799.

[18] Wallach, Hanna M. "Topic modeling: beyond bag-of-words." In *Proceedings of the 23rd international conference on Machine learning*, pp. 977-984. ACM, 2006.

[19] Page, Lawrence, Sergey Brin, Rajeev Motwani, and Terry Winograd. "The PageRank citation ranking: bringing order to the web." (1999).

[20] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS ’15, New York, NY, USA, 2015. ACM.

[21] Armbrust, Michael, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee et al. "A view of cloud computing."*Communications of the ACM* 53, no. 4 (2010): 50-58.

[22] Dreher, Patrick, Chansup Byun, Chris Hill, Vijay Gadepally, Bradley Kuszmaul, and Jeremy Kepner. "PageRank Pipeline Benchmark: Proposal for a Holistic System Benchmark for Big-Data Platforms." In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2016 IEEE International*, IEEE, 2016.

[23] Elmore, A., J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer et al. "A demonstration of the BigDAWG polystore system." *Proceedings of the VLDB Endowment* 8, no. 12 (2015): 1908-1911.